Gof Design Patterns

# Behavioral design patterns

1

**Behavioral Patterns**

Concerned with the interaction between Objects

# Behavioral Patterns



- Describe algorithms, assignment of responsibility, and interactions between objects (behavioral relationships)

General example:

- Model-view-controller in UI application
- Iterating over a collection of objects
- Comparable interface in Java


Template

# Template



- Provide an abstract definition for a method or a class and redefine its behavior later or on the fly without changing its structure.
- Define the skeleton of the algorithm in an operation and deferring the exact implementations of the steps of the algorithms to its subclasses. Template method uses the HR policy of "we will call you" which means the exact implementations of the algorithm will be called by the base class.

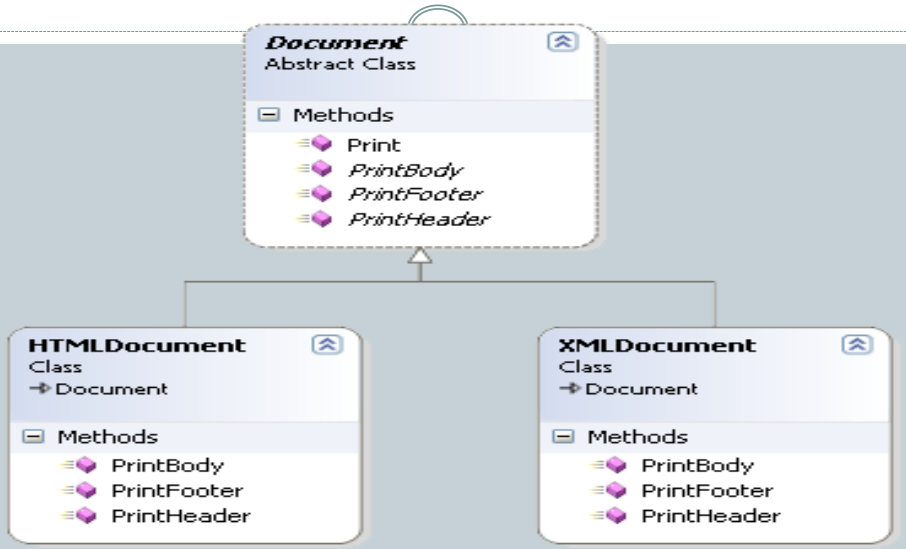


# Applicability



- Use the Template Method pattern
  - To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary
  - When common behavior among subclasses should be factored and localized in a common class to avoid code duplication
  - To control subclasses extensions. You can define a template method that calls “hook” operations at specific points, thereby permitting extensions only at those points.

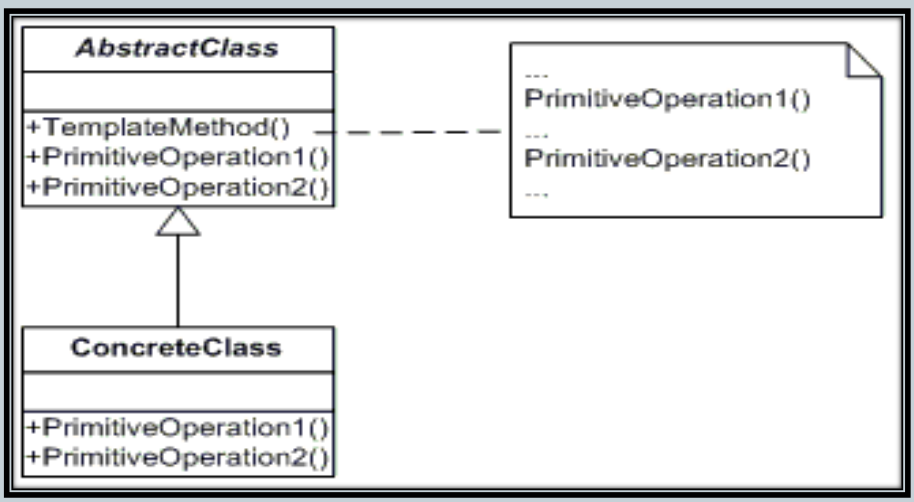
# Example






# Structure

8



# When and how to use Template

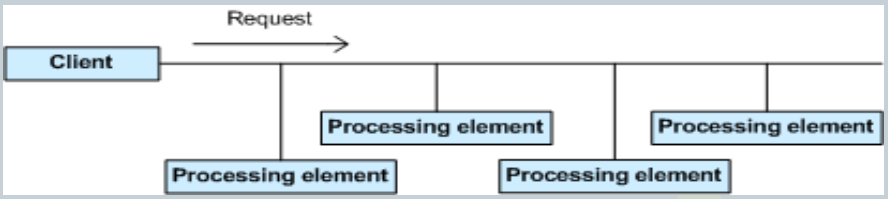


1. Examine the algorithm, and decide which steps are standard and which steps are peculiar to each of the current classes.
2. Define a new abstract base class to host the "don't call us, we'll call you" framework.
3. Move the shell of the algorithm (now called the "template method") and the definition of all standard steps to the new base class.
4. Define a placeholder or "hook" method in the base class for each step that requires many different implementations. This method can host a default implementation – or – it can be defined as abstract (Java) or pure virtual (C++).
5. Invoke the hook method(s) from the template method.
6. Each of the existing classes declares an "is-a" relationship to the new abstract base class.
7. Remove from the existing classes all the implementation details that have been moved to the base class.
8. The only details that will remain in the existing classes will be the implementation details peculiar to each derived class.

***Template Pattern***


## Chain of Responsibility

# Problem Statement – Approval System




| Manager | Vice President | President | Director |
|---|---|---|---|
| 5000 | 25000 | 40000 | 75000 |

*Chain of Responsibility Pattern*

# Sample Code

```
abstract class ApprovalPower {

    protected ApprovalPower successor;

    public void setSuccessor(ApprovalPower successor){
        this.successor = successor;
    }

    abstract public void processRequest(PurchaseRequest request);
}
```

```
class President extends ApprovalPower {
    private final double ALLOWABLE_LIMIT = 40000;

    public void processRequest(PurchaseRequest request){
        if( request.getAmount() < ALLOWABLE_LIMIT )
            System.out.println("President will approve $" + request.getAmount());
        else
            if( successor != null)
                successor.processRequest(request);
    }
```

```
class Director extends ApprovalPower {
    private final double ALLOWABLE_LIMIT = 75000;
    public void processRequest(PurchaseRequest request ) {
        if( request.getAmount() < ALLOWABLE_LIMIT )
            System.out.println("Director will approve $"+ request.getAmount());
        else
            System.out.println( "Your request for $" + request.getAmount() + " needs a board meeting!");
    }
```

*Chain of Responsibility Pattern*

# Code Illustration (..contd)

```
class PurchaseRequest {
    private int requestNumber;
    private double amount;
    private String purpose;
    public PurchaseRequest(int requestNumber, double amount, String purpose){
        this.requestNumber = requestNumber;
        this.amount = amount;
        this.purpose = purpose;
    }
    public double getAmount() {
        return amount;
    }
    public String getPurpose() {
        return purpose;
    }
```

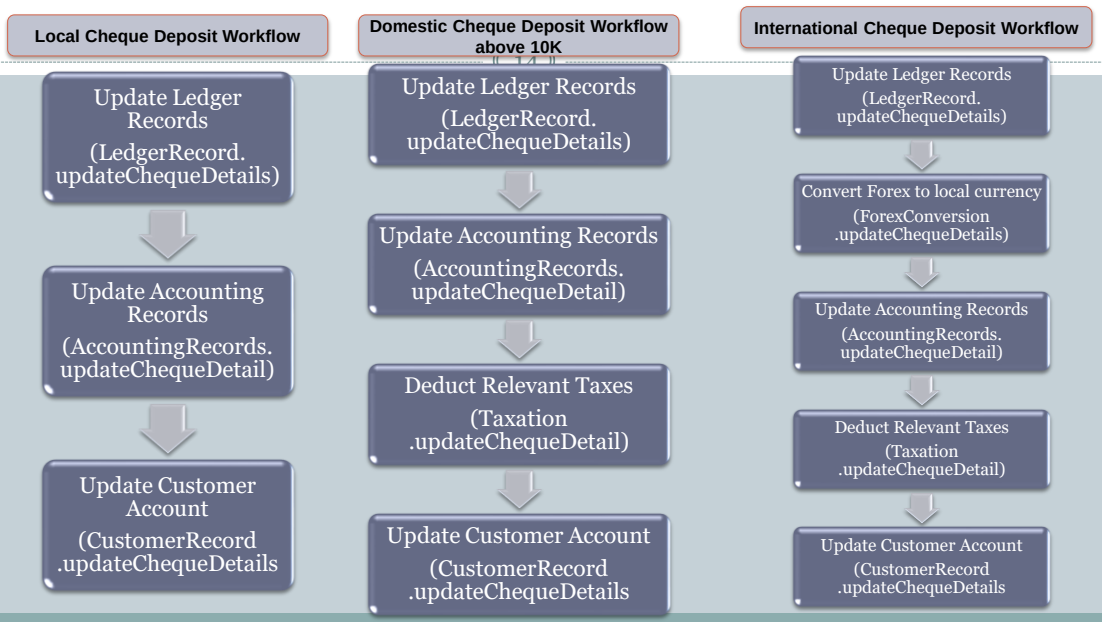
Request object Encapsulated

```
public static void main(String[] args) throws Exception{
    Manager manager = new Manager();
    Director director = new Director();
    VicePresident vp = new VicePresident();
    President president = new President();
    // Link the chain at runtime
    // Manager -> Vice President -> President --> Director
    manager.setSuccessor(vp);
    vp.setSuccessor(president);
    president.setSuccessor(director);

    // always leave the request at the start of the chain
    manager.processRequest(new PurchaseRequest(0, 2000, "Stationery"));
    manager.processRequest(new PurchaseRequest(0, 12000, "Travel"));
    manager.processRequest(new PurchaseRequest(0, 28000, "International Travel"));
```

Client sets the process link list

*Chain of Responsibility Pattern*

# Cheque deposit example




*Chain of Responsibility Pattern*

# Sample Code

```
class ChequeDetails {
    private int chequeValue;
    private String chequeId, micrCode;
    private String toAccountNo, fromAccountNo, fromBankCode, toBranchCode;
    private String currencyCode, countryCode, fromBranchCode;

}
```

Request

```
abstract class TransactionHandler {
    protected TransactionHandler successor;

    public void setSuccessor(TransactionHandler successor) {
        successor = successor;
    }

    public abstract void handleChequeDeposit(ChequeDetails chequeDepositRequest);

    public abstract void handleWithdrawal(ChequeDetails chequeDepositRequest);
}
```

Handler

```
class ForexConversion extends TransactionHandler {
    public void handleChequeDeposit(ChequeDetails chequeDepositRequest) {
        // convert forex to local currency and update ChequeDetails

        if (super.successor != null) {
            super.successor.handleChequeDeposit(chequeDepositRequest);
        }
    }
    @Override
    public void handleWithdrawal(ChequeDetails chequeD
```

ConcreteHandler

*Chain of Responsibility Pattern*

# Sample Code (..contd)

```
class LedgerRecord extends TransactionHandler {
    public void handleChequeDeposit(ChequeDetails chequeDepositRequest) {
        // update Ledger records with Cheque details
        if (super.successor != null) {
            super.successor.handleChequeDeposit(chequeDepositRequest);
        } else {
            // end the chain
        }
    }
```

```
class AccountingRecord extends TransactionHandler {
    public void handleChequeDeposit(ChequeDetails chequeDepositRequest) {
        // update Accounting records with Cheque details
        if (super.successor != null) {
            super.successor.handleChequeDeposit(chequeDepositRequest);
        } else {
            // end the chain
        }
    }
```

```
class CustomerRecord extends TransactionHandler {
    public void handleChequeDeposit(ChequeDetails chequeDepositRequest) {
        // update Ledger records with Cheque details
        if (super.successor != null) {
            super.successor.handleChequeDeposit(chequeDepositRequest);
        } else {
            // end the chain
        }
    }
```

ConcreteHandlers

# Sample Code (..contd)

```
class TaxationFees extends TransactionHandler {
    public void handleChequeDeposit(ChequeDetails chequeDepositRequest) {
        /* if cheque type is domestic and greater than 10K, deduct taxes
         * if cheque type is international, deduct taxes and
         * international transaction fees
         */
        if (super.successor != null) {
            super.successor.handleChequeDeposit(chequeDepositRequest);
        } else {
            // end the chain
    }
```

ConcreteHandler

```
    public static void main(String[] args) {

        ChequeDetails chequeDetails = new ChequeDetails();

        // if domestic cheque -- define the relevant workflow
        ledgerRecordHandler.setSuccessor(accountingRecordHandler);
        accountingRecordHandler.setSuccessor(taxFeesHandler);
        taxFeesHandler.setSuccessor(customerRecordHandler);
        // launch the request at the start of the chain
        ledgerRecordHandler.handleChequeDeposit(chequeDetails);

        // if international cheque -- define the relevant workflow
        ledgerRecordHandler.setSuccessor(forexConversionHandler);
        forexConversionHandler.setSuccessor(accountingRecordHandler);
        accountingRecordHandler.setSuccessor(taxFeesHandler);
        taxFeesHandler.setSuccessor(customerRecordHandler);
        // launch the request at the start of the chain
        ledgerRecordHandler.handleChequeDeposit(chequeDetails);
    }
```

Client

# Chain of Responsibility

18

## Intent

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
- An object-oriented linked list with recursive traversal

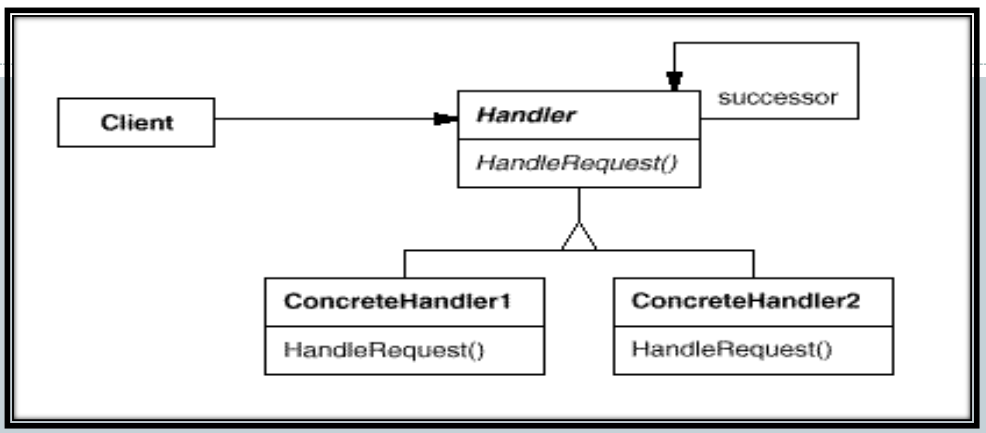
# Benefits



- One request could be handled by more than one object.
- Don't know which object should handle a request, probably more than one object will handle it automatically.
- Reduce coupling.
- Flexible in handling a request



# Structure




- **Handler -** defines an interface for handling requests
  - **ConcreteHandler** - handles the requests it is responsible for If it can handle the request it does so, otherwise it sends the request to its successor
  - **Client** - sends commands to the first object in the chain that may handle the command

## Checklist for using CoR

21

- The base class maintains a “next” pointer.
- Each derived class implements its contribution for handling the request.
- If the request needs to be “passed on”, then the derived class “calls back” to the base class, which delegates to the “next” pointer.
- The client (or some third party) creates and links the chain (which may include a link from the last node to the root node).
- The client “launches and leaves” each request with the root of the chain.
- Recursive delegation produces the illusion of magic.

## Applicability

22

Use the Chain of Responsibility when

- more than one object may handle a request and the handler is not known a priori.
- you want to issue a request to one of several objects without specifying the receiver explicitly.
- the set of objects that can handle a request should be specified dynamically.

## Consequences

23

- Reduced coupling : Neither the sender nor the receiver have an explicit knowledge of each other. An object only knows that a request will be appropriately handled, but does not know that the structure of the chain. An object only needs to know its immediate successor. This can simplify object interconnections.
- Flexibility in changing responsibilities. You can add or change responsibilities for handling requests dynamically
- Receipt is not guaranteed. A request may not be handled by any object, and may "just fall off the end".

*Chain of Responsibility Pattern*

Command

# Command pattern



## Problem

- Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

## Intent

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Promote "invocation of a method on an object" to update object status
- An object-oriented callback
- allows saving the requests in a queue
- decouples the object that invokes the action from the object that performs the action. Due to this usage it is also known as Producer - Consumer design pattern.

*Command Pattern*

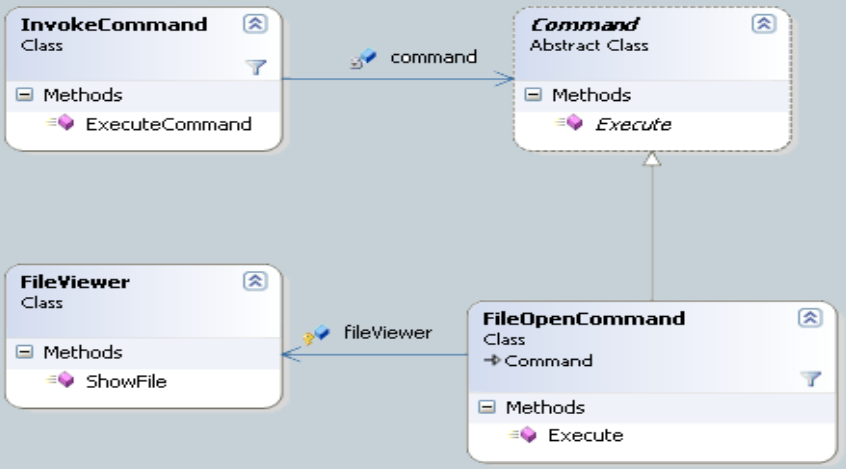
# Command



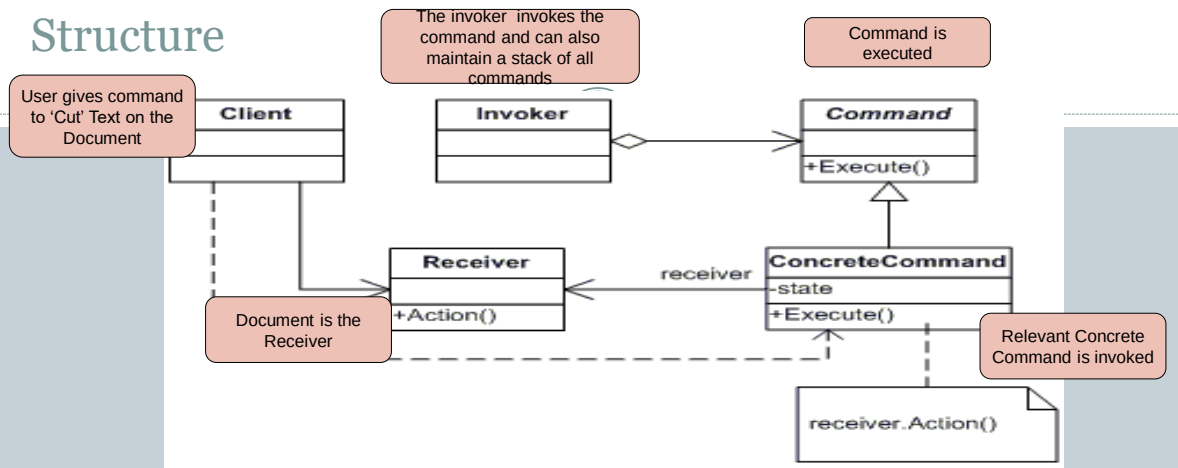
- Command decouples the object that invokes the operation from the one that knows how to perform it.
  - The receiver is the one who knows how to perform the operations needed, the purpose of the command being to help the client to delegate its request quickly and to make sure the command ends up where it should

# File Open Command

27




# Structure




- **Command** - declares an interface for executing an operation;
- **ConcreteCommand** - extends the Command interface, implementing the Execute method by invoking the corresponding operations on Receiver. It defines a link between the Receiver and the action.
- **Client** - creates a ConcreteCommand object and sets its receiver;
- **Invoker** - asks the command to carry out the request;
- **Receiver** - knows how to perform the operations;

# Command Pattern Sample Code

```
// Command
abstract class TextCommand
{
    public abstract void execute();
}
abstract class UndoableTextCommand extends TextCommand
{
    public abstract void undo();
}
```

```
// Concrete Command
class BoldCommand extends UndoableText
{
    private Document document;
    // Receiver is passed to the comma
    public BoldCommand(Document doc)
    {
        this.document = doc;
    }
    @Override
    public  void execute()
    {
        // make the text bold
    }

    @Override
    public  void undo()
    {
        // remove the bold from the te
    }
}
```

```
//Concrete Command
class CutCommand extends UndoableTextCommand
{
    private Document document;
    private String previousText;
    public CutCommand(Document doc)
    {
        this.document = doc;
    }
    @Override
    public  void execute()
    {
        // Will cut the text from the document
    }

    @Override
    public  void undo()
    {
        // Will undo the cut command
    }
}
```

# Command Pattern Sample Code (Contd..)

```
// Receiver
public class Document{
    private String text;
    public Document() {

    }
    public String getText() {
        return text;
    }

    publi
        t
    }
}
```

```
// Invoker
class CommandManager
{
    private Stack commandStack = new Stack();

    public void executeCommand(TextCommand command)
    {
        command.execute();
        if (command instanceof UndoableTextCommand)
        {
            commandStack.push(command);
        }
    }

    public void undo()
    {
        if (commandStack.size() > 0)
        {
            UndoableTextCommand command = (UndoableTextCommand)commandStack.pop();
            command.undo();
        }
    }
}
```

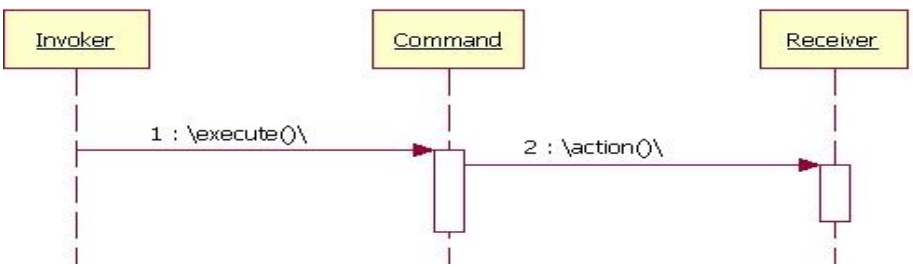
# Command Pattern Sample Code (Contd..)



```
// Client
class DocumentClient
{
    public static void main(String[] args) {
        Document document = new Document();
        CommandManager commandManager = new CommandManager();
        commandManager.executeCommand(new CutCommand(document));
        commandManager.executeCommand(new PasteCommand(document));
        commandManager.executeCommand(new BoldCommand(document));
        commandManager.undo();
        commandManager.executeCommand(new PasteCommand(document));
        commandManager.executeCommand(new CutCommand(document));
        commandManager.executeCommand(new PasteCommand(document));
        commandManager.executeCommand(new BoldCommand(document));
        commandManager.undo();
        commandManager.undo();
        commandManager.undo();

    }
}
```

# Sequence Diagram





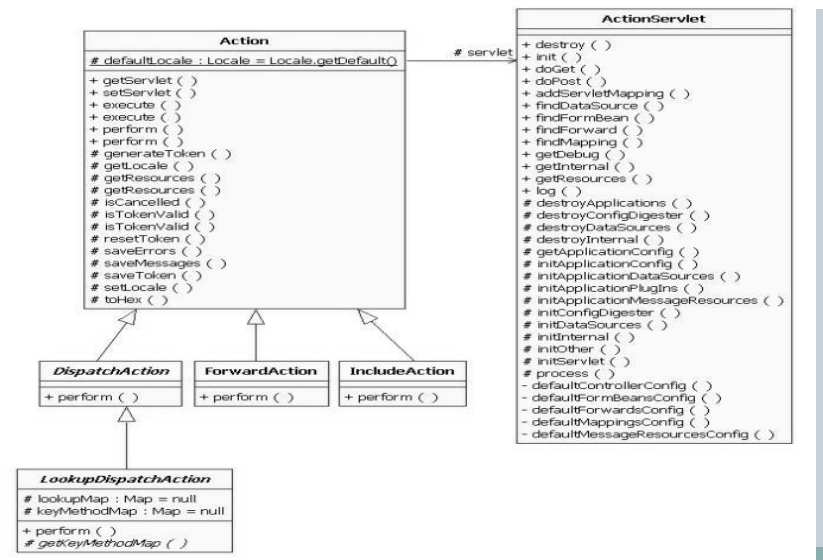
# Checklist for using command pattern

33

- Define a Command interface with a method signature like execute().
- Create one or more derived classes that encapsulate some subset of the following: a "receiver" object, the method to invoke, the arguments to pass.
- Instantiate a Command object for each deferred execution request.
- Pass the Command object from the creator (aka sender) to the invoker (aka receiver).
- The invoker decides when to execute().



# Command and Struts



## Common uses

35

- Multi level undo & redo
- Transactional behavior
- GUI buttons and menu items
- Progress bars
- Calculator, Text editors etc



## Consequences

36

- Decouples the object that invokes an operation from the one that executes it.
- Command objects can be manipulated and extended like any other object.
- You can assemble commands into a composite command, to carry out a series of operations
- You can easily add new commands without changing existing classes.

# Applicability

37

Use the command pattern

- To encapsulate action requests. In procedural languages, you can use callback functions for this purpose. The command pattern is an object oriented way of doing the same thing
- When you want to support undo or redo operations
- When you need to support logging changes so that they can be reapplied in case of a system crash

Observer

# Observer

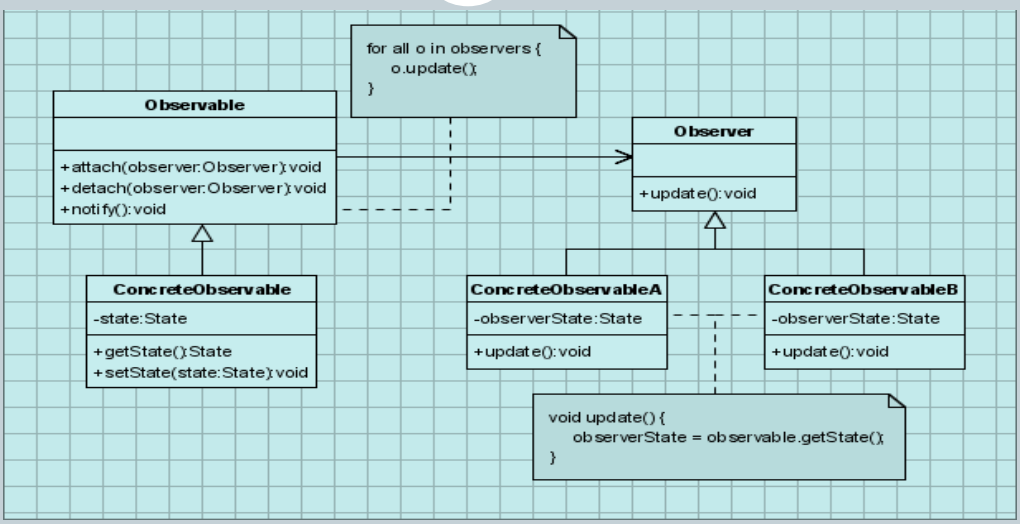
39

- Define a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Also known as Publish / Subscribe



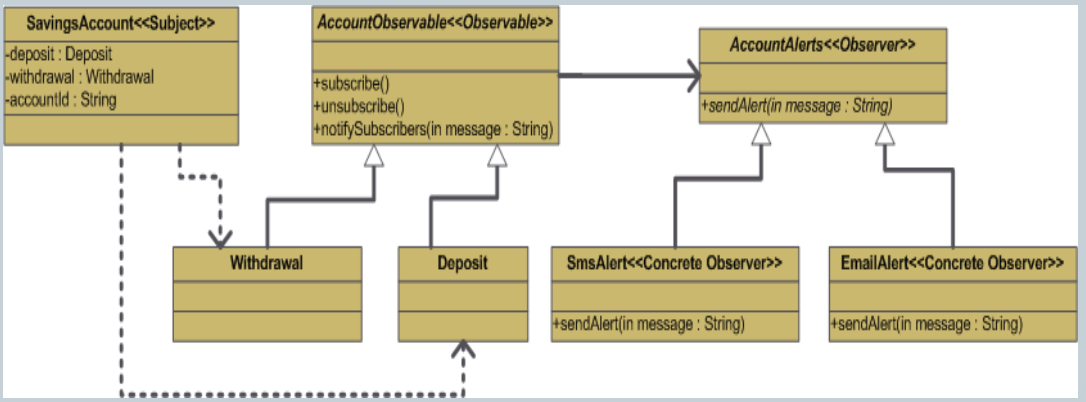
# Structure

40



# Example : Account Alerts

41




# Sample Code : Account Alerts

42

```
// subject being observed
class SavingsAccount
{
    private Withdrawal withdrawal;
    private Deposit deposit;
    private int currentBalance;
    private String bankAccountId;
    public SavingsAccount(String bankAccountId) {
        this.bankAccountId = bankAccountId;
        withdrawal = new Withdrawal();
        deposit = new Deposit();
    }
    public void subscribeForDepositAlerts(AccountObserver observer) {
        deposit.subscribe(observer);
    }
    public void subscribeForWithdrawalAlerts(AccountObserver observer)  {
        withdrawal.subscribe(observer);
    }

    public void deposit(int depositAmt) {
        deposit.depositIntoAct(depositAmt);
    }
    public void withdrawal(int withdrawAmt) {
        withdrawal.withdrawalFromAct(withdrawAmt);
    }
```

# .. Code contd.. Account Alerts

```
// Observable - List of event classes which are interested in particular event
class AccountObservable {
    ArrayList<AccountObserver> observers = new ArrayList<AccountObserver>();

    public void subscribe(AccountObserver observer) {
        observers.add(observer);
    }

    public void unSubscribe(AccountObserver observer) {
        observers.remove(observer);
    }

    public void notifySubscribers(String message) {
        System.out.println("******* Notify Subscribers called ********");
        // notify all the subscribers
        fo
        }
    }
}
```

```
//Whenever a debit occurs on an account more than a specific amount
private class Withdrawal extends AccountObservable {
    public void withdrawalFromAct(int withdrawAmt)
    {
        String message = bankAccountId + " : Withdrawal amount : "+withdrawAmt;
        System.out.println("\n ######## "+message);
        notifySubscribers(message);
    }
}
// Whenever a credit occurs on an account more than a specific amount
private class Deposit extends AccountObservable {
    public void depositIntoAct(int depositAmt)
    {
        String message = bankAccountId +" : Depositing amount : "+depositAmt;
        System.out.println("\n ######## "+message);
        notifySubscribers(message);
    }
}
```

# .. Code contd.. Account Alerts



```
abstract class AccountObserver {
    public abstract void sendAlert(String message);
}

// Concrete Observers
class SMSAlert extends AccountObserver {
    @Override
    public void sendAlert(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

//Concrete Observers
class EmailAlert extends AccountObserver {
    @Override
    public void sendAlert(String message) {
        System.out.println("Sending Email: " + message);
    }

}
```

# .. Code Contd

Subscribing for the alerts

```
blic static void main(String[] args) {
  SavingsAccount sa1 = new SavingsAccount("SA01234");
  SavingsAccount sa2 = new SavingsAccount("SA02345");
  SavingsAccount sa3 = new SavingsAccount("SA04567");
  SavingsAccount sa4 = new SavingsAccount("SA06789");
  sa1.subscribeForDepositAlerts(new SMSAlert());
  sa1.subscribeForDepositAlerts(new EmailAlert());
  sa1.subscribeForWithdrawalAlerts(new SMSAlert());
  sa1.subscribeForWithdrawalAlerts(new EmailAlert());
  sa2.subscribeForDepositAlerts(new SMSAlert());
  sa2.subscribeForDepositAlerts(new EmailAlert());
  sa3.subscribeForWithdrawalAlerts(new SMSAlert());
  sa3.subscribeForWithdrawalAlerts(new EmailAlert());
  sa4.subscribeForDepositAlerts(new SMSAlert
  sa4.subscribeForWithdrawalAlerts(new SMSAl
```

Client transactions

```
sa1.deposit(5000);
sa1.withdrawal(7000);
sa1.deposit(8000);

sa2.deposit(10000);
sa2.withdrawal(12000);

sa3.deposit(15000);
sa3.withdrawal(18000);

sa4.deposit(20000);
sa4.withdrawal(25000);
```

```
 ######## SA04567 : Depositing amount : 15000
****** Notify Subscribers called *******

 ######## SA04567 : Withdrawal amount : 18000
****** Notify Subscribers called *******
Sending SMS: SA04567 : Withdrawal amount : 18000
Sending Email: SA04567 : Withdrawal amount : 18000

 ######## SA06789 : Depositing amount : 20000
****** Notify Subscribers called *******
Sending SMS: SA06789 : Depositing amount : 20000

 ######## SA06789 : Withdrawal amount : 25000
****** Notify Subscribers called *******
Sending SMS: SA06789 : Withdrawal amount : 2
```

```
 ######## SA01234 : Depositing amount : 5000
****** Notify Subscribers called *******
Sending SMS: SA01234 : Depositing amount : 5000
Sending Email: SA01234 : Depositing amount : 5000

 ######## SA01234 : Withdrawal amount : 7000
****** Notify Subscribers called *******
Sending SMS: SA01234 : Withdrawal amount : 7000
Sending Email: SA01234 : Withdrawal amount : 7000

 ######## SA01234 : Depositing amount : 8000
****** Notify Subscribers called *******
Sending SMS: SA01234 : Depositing amount : 8000
Sending Email: SA01234 : Depositing amount : 8000

 ######## SA02345 : Depositing amount : 10000
****** Notify Subscribers called *******
Sending SMS: SA02345 : Depositing amount : 10000
Sending Email: SA02345 : Depositing amount : 10000

 ######## SA02345 : Withdrawal amount : 12000
****** Notify Subscribers called *******
```

Output. Alerts sent to the customers based on subscription



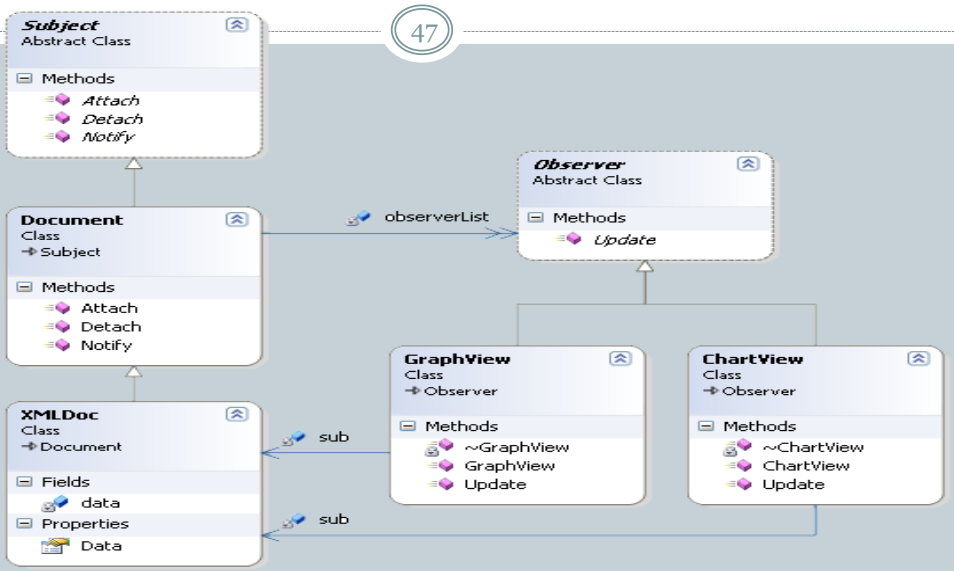
# How to create an Observer Pattern

46

- Differentiate between the core (or independent) functionality and the optional (or dependent) functionality.
- Model the independent functionality with a “subject” abstraction.
- Model the dependent functionality with an “observer” hierarchy.
- The Subject is coupled only to the Observer base class.
- The client configures the number and type of Observers.
- Observers register themselves with the Subject.
- The Subject broadcasts events to all registered Observers.
- The Subject may “push” information at the Observers, or, the Observers may “pull” the information they need from the Subject.

## Example : Updating Charts and graph based on XML data updates







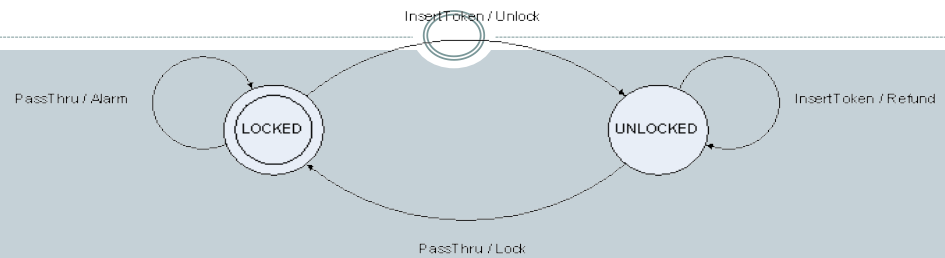

## Observer pattern used in Java & C#



- Java
  - java.util.Observer / java.util.Observable
  - All implementations of java.util.EventListener
  - javax.servlet.http.HttpSessionBindingListener
  - javax.servlet.http.HttpSessionAttributeListener

# State

## Turnstile example




**Code would look like**

```
public void passThru()
{
    If(currentState == 1) {   // if locked
            currentState == 2; // alarm state
            raiseAlarm();
    }
    If(currentState == 0) { // if unlocked
            currentState == 1;
    }
}
.....
```

```
public void insertToken()
{
    If(currentState == 1)  {   // if locked
            currentState == 0; //unlock
            unlockTurnstile();
    }
    If(currentState == 0)  {  // if unlocked
            // no change to current state
            allowRefund();
    }
}
.....
```

*State Pattern*

# Sample Code



```
public class Turnstile {
    TurnstileState currentState = TurnstileState.lockedState;

    public void insertToken()
    {
        currentState.insertToken(this);
        System.out.println("Current State after inserting Token: "+ currentState.getClass());
    }
    public void passThru()
    {
        currentState.passThru(this);
        System.out.println("Current State after Passing through: "+ currentState.getClass());
    }
```

```
interface TurnstileState
{
    public static final LockedState lockedState = new LockedState();
    public static final UnLockedState unlockedState = new UnLockedState();
    public void passThru(Turnstile turnstile);
    public void insertToken(Turnstile turnstile);
    public String getState();

}
```



# Sample Code

```
class LockedState implements TurnstileState
{
    @Override
    public void insertToken(Turnstile turnstile) {
        System.out.println("**Locked State** Insert Token : Change State");
        turnstile.currentState = unlockedState;
    }
    @Override
    public void passThru(Turnstile turnstile) {
        // raiseAlarm();
        System.out.println("**Locked State** Pass Thru : No Change State");
        System.out.println("** RAISE ALARM **");
        turnstile.currentState =  lockedState;
```

```
class UnLockedState implements TurnstileState
{
    @Override
    public void insertToken(Turnstile turnstile) {
        System.out.println("**UnLocked State** InsertToken : No Change State");
        turnstile.currentState =  unlockedState;
    }
    @Override
    public void passThru(Turnstile turnstile) {
        System.out.println("**UnLocked State** Pass Thru : Change State");
        turnstile.currentState =  lockedState;
    }
    public String getState(){
        return "UnLocked State";
    }
```

# Sample Code (..contd)



```
public static void main(String[] args) {
    Turnstile t = new Turnstile();
    t.insertToken();
    t.passThru();
    t.insertToken();
    t.passThru();
    t.insertToken();
    t.insertToken();
    t.passThru();
    t.passThru();
    t.insertToken();
}
```

Problems | Tasks | Properties | Servers | Data Source Explorer | Snippets | Console

<terminated> Turnstile [Java Application] C:\FAST\DeveloperDesktop\Tools\JDK\jre6\bin\javaw.exe (Nov 4, 2011 12:26:24 PM)

```
**Locked State** Insert Token : Change State
Current State after inserting Token: class com.jp.behavioral.state.UnLockedState
**UnLocked State** Pass Thru : Change State
Current State after Passing through: class com.jp.behavioral.state.LockedState
**Locked State** Insert Token : Change State
Current State after inserting Token: class com.jp.behavioral.state.UnLockedState
**UnLocked State** Pass Thru : Change State
Current State after Passing through: class com.jp.behavioral.state.LockedState
**Locked State** Insert Token : Change State
Current State after inserting Token: class com.jp.behavioral.state.UnLockedState
**UnLocked State** InsertToken : No Change State
Current State after inserting Token: class com.jp.behavioral.state.UnLockedState
**UnLocked State** Pass Thru : Change State
Current State after Passing through: class com.jp.behavioral.state.LockedState
**Locked State** Pass Thru : No Change State
** RAISE ALARM **
Current State after Passing through: class com.jp.behavioral.state.LockedState
**Locked State** Insert Token : Change State
Current State after inserting Token: class com.jp.behavioral.state.UnLockedState
```



# State Pattern



## Intent

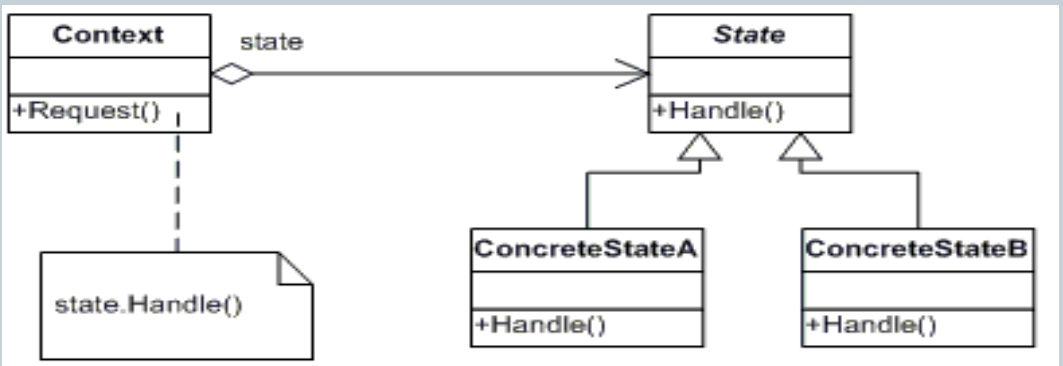
- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- An object-oriented state machine
- wrapper + polymorphic wrappee + collaboration

## State Pattern -

- The State pattern is a solution to the problem of how to make behavior depend on state.

# Structure

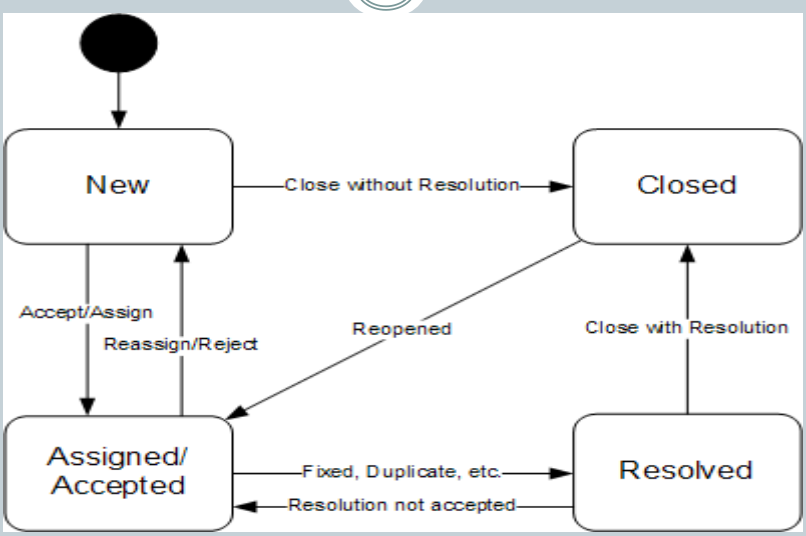
55






# Software defect example

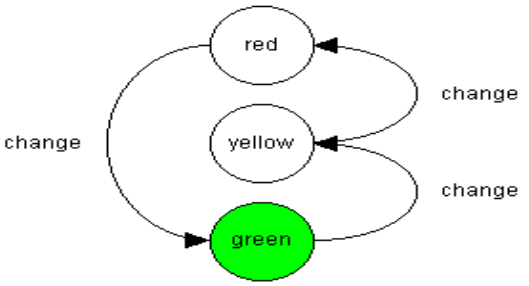
56



# Traffic light example

57

```
class TrafficLight {

    private ITrafficLightState state;

    public void change() {
        state.change(this);
    }

    public void ReportState() {
        state.ReportState();
    }
```




```
interface ITrafficLightState {
    public static final RedLight redLight = new RedLight();
    public static final YellowLight yellowLight = new YellowLight();
    public static final GreenLight greenLight = new GreenLight();
    // It takes the existing light and changes it.
    void change(TrafficLight light);
    // We'll display the color it's on.
    void reportState();
}
```

# Traffic light Code Contd.

```
class GreenLight implements ITrafficLightState {
    public void change(TrafficLight light) {
        light.setState(yellowLight);
    }
    public void reportState() {
        System.out.println("Green Light");
    }
}

class YellowLight implements ITrafficLightState {
    public void change(TrafficLight light) {
        System.out.println("Capture view on camera just before changing state to Red");
        light.setState(redLight);
    }
    public void reportState() {
        System.out.println("Yellow Light");
    }
}

class RedLight implements ITrafficLightState {
    public void change(TrafficLight light) {
        light.setState(greenLight);
    }
    public void reportState() {
        System.out.println("Red Light");
    }
}
```

# Consequences

59

- Localizes state-specific behavior and partitions behavior for different states. New states and transitions can be added easily by defining new subclasses.
- Makes state transitions explicit
- State objects can be shared
- Makes code easier to read and maintain.

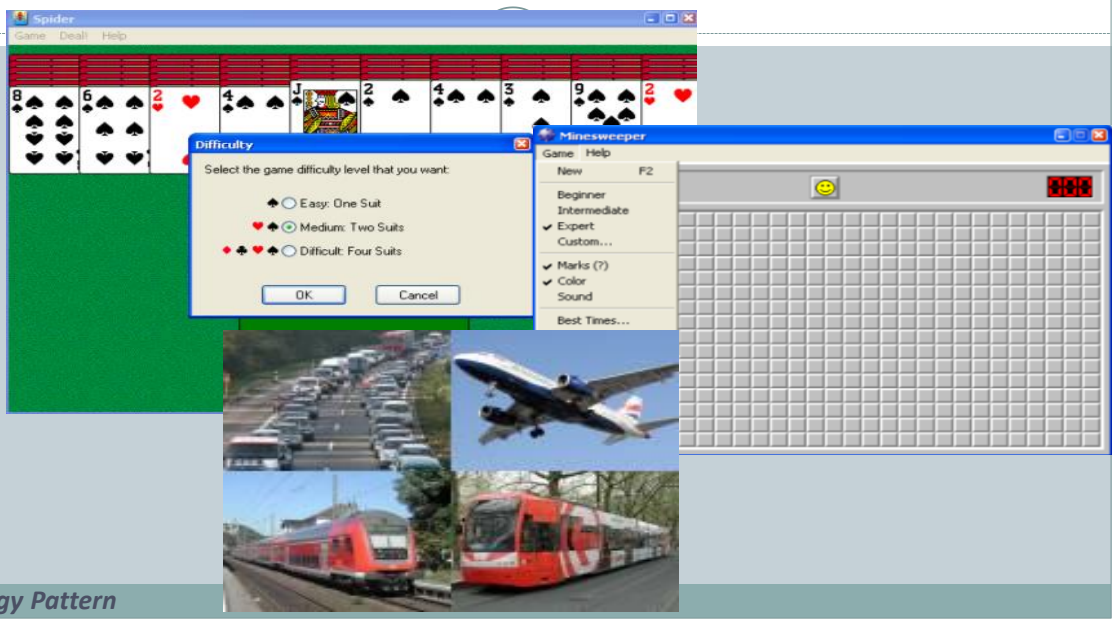
**Useful Tip :**

- State objects can often be implemented as Singletons.

*State Pattern*

Strategy

# Game Strategy






# Strategy Pattern



**When to use?**

- Do we have a varying rule or algorithm

**How to use?**

- Define a family of algorithms, encapsulate each one and make them interchangeable. It allows us to change the algorithm independently with out changing the client using it.
- Capture the abstraction in an interface, bury implementation details in derived classes.

# Benefits

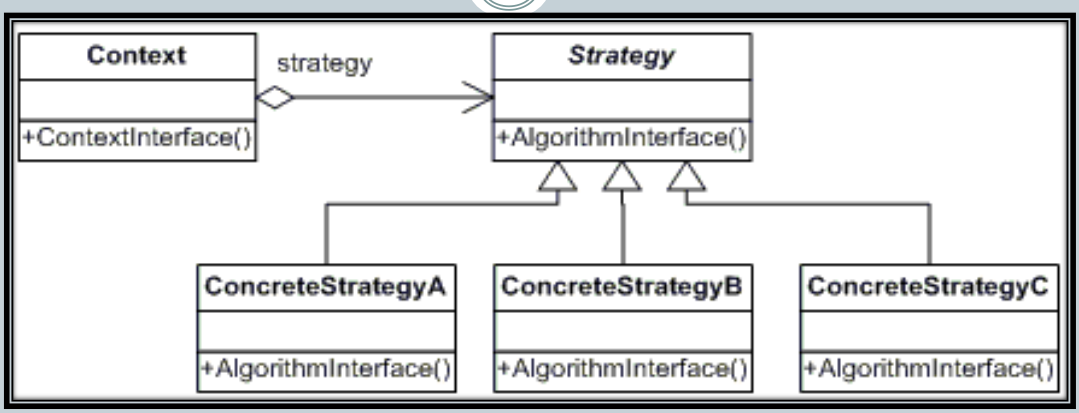
63

- Encapsulate various algorithms to do more or less the same thing.
- Need one of several algorithms dynamically.
- The algorithms are exchangeable and vary independently
- Configure a class with one of many related classes (behaviors).
- Avoid exposing complex and algorithm-specific structures.
- Data is transparent to the clients.
- Reduce multiple conditional statements.
- Provide an alternative to subclassing.



# Structure

64



# Strategy Example : Code

```
// Configured with a ConcreteStrategy object and maintains a reference to a
// Strategy object
class Context {
    private Strategy strategy;
    // Constructor
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }
    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}
```

```
//The classes that implement a concrete strategy should implement this
//The context class uses this to call the concrete strategy
interface Strategy {
    int execute(int a, int b);
}
```

```
// Implements the algorithm using the strategy interface
class ConcreteStrategyAdd implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyAdd's execute()");
        return a + b; // Do an addition with a and b
    }
}
```

```
class ConcreteStrategySubtract implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategySubtract's execute()");
        return a - b; // Do a subtraction with a and b
    }
}
```

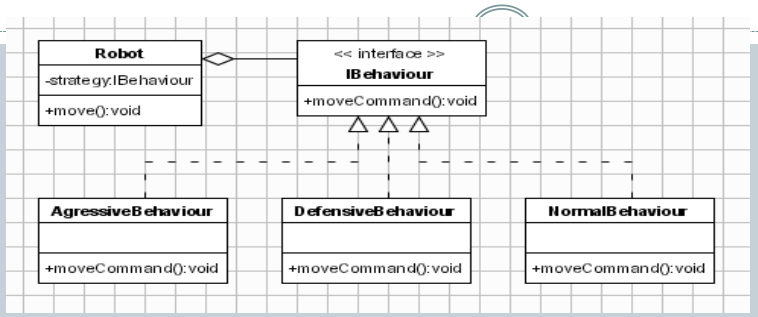
```
class ConcreteStrategyMultiply implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyMultiply's execute()");
        return a * b; // Do a multiplication with a and b
    }
}
```

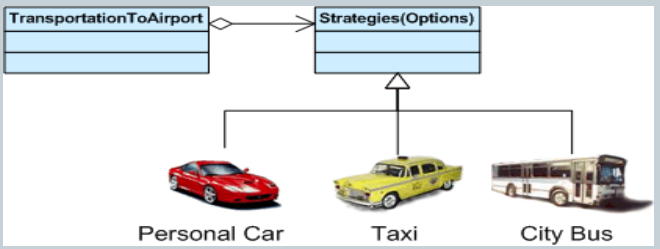
# ..code contd

66

```
// StrategyExample test application
class StrategyExample {

    public static void main(String[] args) {
        Context context;
        // Three contexts following different strategies
        context = new Context(new ConcreteStrategyAdd());
        int resultA = context.executeStrategy(3, 4);

        context = new Context(new ConcreteStrategySubtract());
        int resultB = context.executeStrategy(3, 4);

        context = new Context(new ConcreteStrategyMultiply());
        int resultC = context.executeStrategy(3, 4);
    }
}
```

# More Examples




# Strategy pattern in Java & C#



- java.util.Comparator. compare method is used in Collections.sort()

  The sort method has the iteration logic, however depending on the comparator provided it, compares the two objects to sort them. Hence the sorting criteria can be changed dynamically without changing the iteration code.

- System.Collections.Generic . List<T> contains the method public void Sort(IComparer<T> comparer)

# Applicability

69

- Use the strategy pattern when
  - A class defines many alternative behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own strategy classes
  - You need different variants of an algorithm.



# State v/s Strategy

70

- State is intrinsic to an object. Clients don't usually set the state of an object. An object changes its own state (with the help of State classes) based on certain events.

  Strategy is usually selected by a client
- State of an object may change often, in response to events.

  Strategy does not change often.

# Memento

## Problem statement



```
public class SavingsAccount{
   //private attributes
      public void deposit (float amt)
         {
          . . . . .
      }
      public void withdraw(float amt)
         {
          . . . . .
      }
      public float getBalance() {
          . . . . .
      }
      . . . .
}
```

```
public class SAController{

      public void process(SavingsAccount
         sa) {
    float bal= sa.getBalance();
    sa.deposit(amt);
    boolean isTransferSuccessful =
    TransferFunds.transfer(sa);
        if(!isTransferSuccessful)
         {
            /// revert to original state
               // how to implement?
        }
        . . . . .
      }
      . . . .
}
```

# Option 1

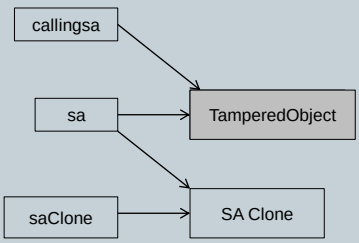


Clone the SavingsAccount object before executing and revert to the cloned object while unexecuting.

Drawbacks

- Any other client might be holding reference to the old object.
- The controller class can tamper the state by setting other values

```
public class SAController{
  public void process(SavingsAccount sa) {
    SavingsAccount saClone = sa.clone();
    boolean isTransferSuccessful =
    TransferFunds.transfer(sa);
     if(!isTransferSuccessful)
      {
          sa = saClone();
      }
  Public class TransferFunds
```






# Option 2



Copy attribute values from SA object to its snapshot, and then in the unexecute operation copy them back

Drawbacks

- All attributes might not be accessible through public accessor functions
- Controller class can still tamper the state by setting the values

# Memento

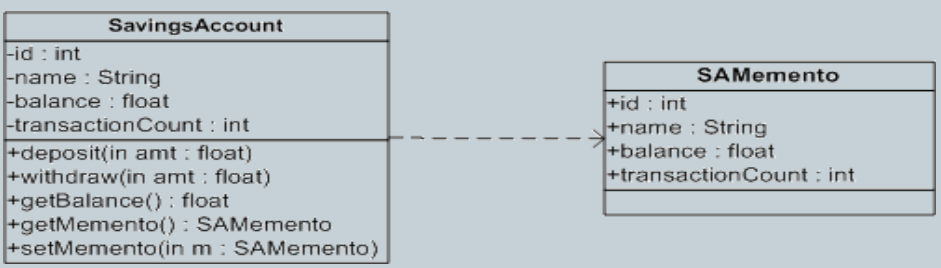
75

- me·men·to : A reminder of the past; a keepsake
  - something that reminds one of past events; souvenir
- The memento design pattern allows you to save historical states of an object and restore the object back from the historical states.
- You are able to do this without breaking encapsulation of the state of the object.



# Memento Approach

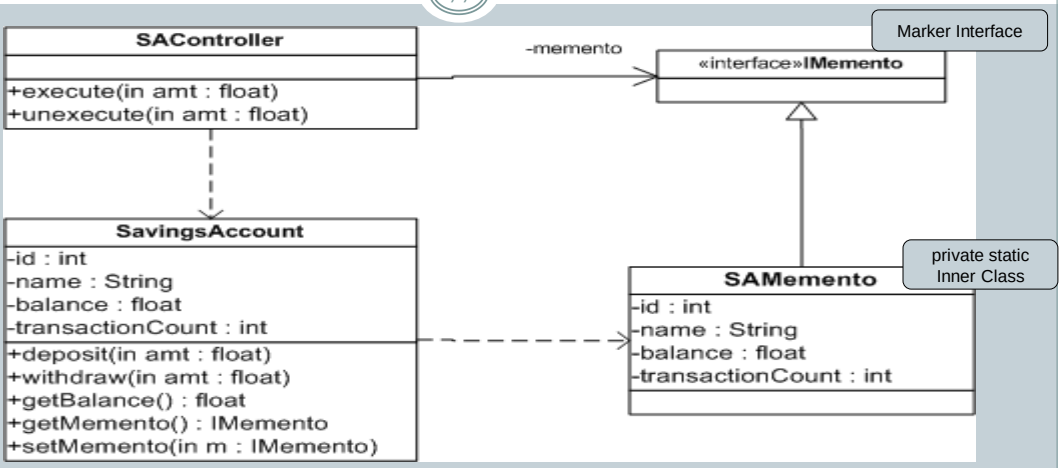
76




Does not break encapsulation.

# Complete Solution

77




*Memento Pattern*

# Memento Pattern Sample Code

```
interface IMemento{

}
```

```
public class Originator {
    private String state;

    // The class could also contain additional data that is not part of the
    // state saved in the memento.
    public void set(String state) {
        System.out.println("Originator: Setting state to " + state);
        this.state = state;
    }
    public IMemento saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(state);
    }
    public void restoreFromMemento(IMemento memento) {
        state = ((Memento)memento).getSavedState();
        System.out.println("Originator: State after restoring from Memento: " + state);
    }
    //private INNER CLASS
    private static class Memento implements IMemento{
        private final String state;

        private Memento(String stateToSave) {
            state = stateToSave;
        }

        private String getSavedState() {
            return state;
        }
    }
}
```

Methods to save and restore from Memento

Has the same attributes as Originator

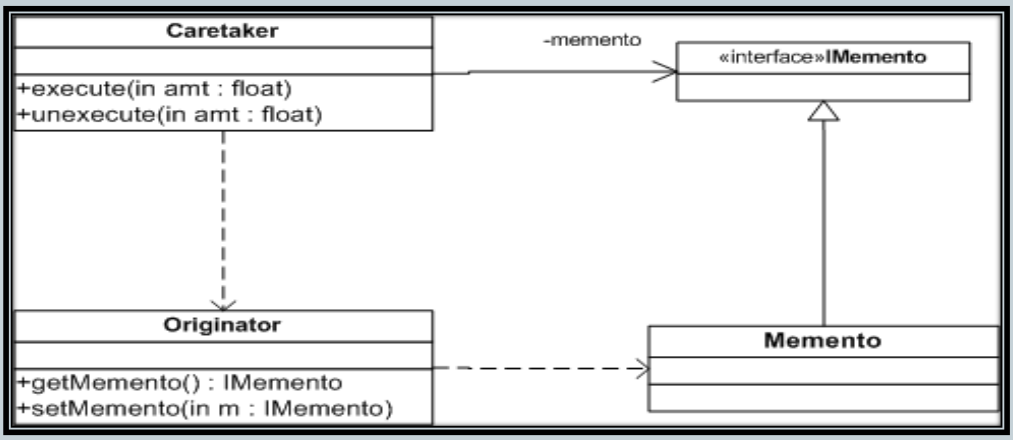
# Memento Pattern Sample Code

```
class Caretaker {
    public static void main(String[] args) {
        Stack<IMemento> savedStates = new Stack<IMemento>();

        Originator originator = new Originator();
        originator.set("State 22");
        originator.set("State 23");
        savedStates.push(originator.saveToMemento());
        originator.set("State 33");
        originator.set("State 41");
        // We can request multiple mementos, and choose which one to roll back to.
        savedStates.push(originator.saveToMemento());
        originator.set("State 43");
        originator.set("State 46");

        originator.restoreFromMemento(savedStates.pop());
    }
}
```

```
<terminated> Caretaker [Java Application] C:\FAST\DeveloperDesktop\Tools\JDK\jre6\bin\javaw.exe (Nov 1, 2011 7:5
Originator: Setting state to State 22
Originator: Setting state to State 23
Originator: Saving to Memento.
Originator: Setting state to State 33
Originator: Setting state to State 41
Originator: Saving to Memento.
Originator: Setting state to State 43
Originator: Setting state to State 46
Originator: State after restoring from Memento: State 41
```

# Structure

80




*Memento Pattern*

## Consequences



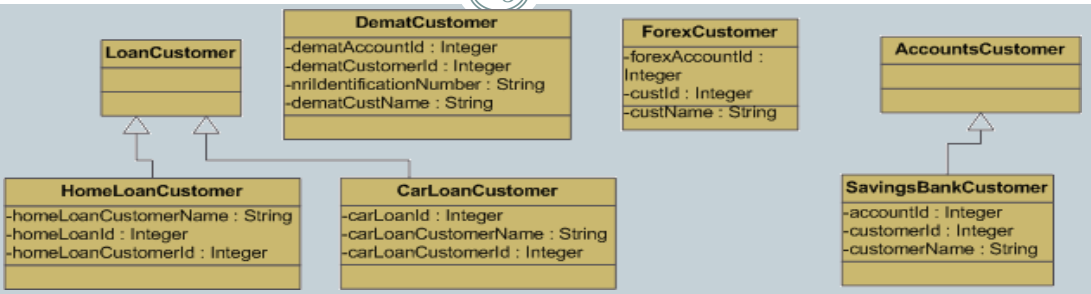
- Preserves encapsulation boundaries
- Simplifies the originator, as it is does not have to manage old state internally
- Considerable overheads may be involved if the originator must copy large amounts of information to store in the memento
- A caretaker is responsible for deleting a memento. However, it does not know how large a memento may be. Thus, an otherwise lightweight caretaker may incur large storage costs when it stores mementos.




Visitor

# Problem Statement






Additional ad hoc requirements :

- Print photo identity card
- Print new year greeting card
- Print discount coupons
- etc



# Code before modification

```
class HomeLoanCustomer {
    private String homeLoanCustomerName;
    private String homeLoanId;
    private int homeLoanCustomerId;
    private String branchCode;
    // other related attributes & methods of Home Loan Customer
}
```

```
class SavingsBankAccountCustomer {
    private String customerName;
    private String bankAccountId;
    private int customerId;
    private String branchCode;
    // other related attributes & methods of Savings Bank Account Customer
}
```

```
class DematCustomer {
    private String dematCustomerName;
    private String dematAccountId;
    private int dematCustomerId;
    private String branchCode;
    // other related attributes & methods related to Demat Customer
}
```

```
class ForexTradingCustomer {
    private String customerName;
    private String forexTradingActId;
    private int forexTradingCustomerId;
    // other related attributes & methods of Forex Trading
}
```

# Step 1: Create Visitor interface and classes

```
interface CustomerVisitor
{
    public void visitHomeLoanCustomer(HomeLoanCustomer hlCustomer);
    public void visitSavingsBankAccountCustomer(SavingsBankAccountCustomer sbaCustomer);
    public void visitDematCustomer(DematCustomer dematCustomer);
    public void visitForexTradingCustomer(ForexTradingCustomer fxCustomer);
}
```

```
class PrintPhotoId implements CustomerVisitor
{
    public void visitHomeLoanCustomer(HomeLoanCustomer hlCustomer){
        // prints photo id for HomeLoanCustomer
    }
}
```

```
class PrintNewYearGreetings implements CustomerVisitor
{
    public void visitHomeLoanCustomer(HomeLoanCustomer hlCustomer){
        // prints new year greetings for HomeLoanCustomer
    }
}
```

```
class PrintDiscountCoupons implements CustomerVisitor
{
    public void visitHomeLoanCustomer(HomeLoanCustomer hlCustomer){
        // prints discount coupons for HomeLoanCustomer
    }
    public void visitSavingsBankAccountCustomer(SavingsBankAccountCustomer sbaCustom
        // prints discount coupons for SavingsBankAccountCustomer
    }
    public void visitDematCustomer(DematCustomer dematCustomer){
        // prints discount coupons for DematCustomer
    }
    public void visitForexTradingCustomer(ForexTradingCustomer fxCustomer){
        // prints discount coupons for ForexTradingCustomer
    }
}
```



## Option A : Step 2 → Call visitor function for each customer



```
List<CustomerVisitor> visitors = new ArrayList<CustomerVisitor>();
visitors.add(new PrintPhotoId());
visitors.add(new PrintNewYearGreetings());
visitors.add(new PrintDiscountCoupons());
for (CustomerVisitor visitor : visitors) {
    // for each visitor, do the action for all customers
    for (DematCustomer dematCustomer : dematCustomers) {
        visitor.visitDematCustomer(dematCustomer);
    }
    for (HomeLoanCustomer homeLoanCustomer : homeLoanCustomers) {
        visitor.visitHomeLoanCustomer(homeLoanCustomer);
    }
    for (SavingsBankAccountCustomer sbCustomer : savingsBankActCustomers) {
        visitor.visitSavingsBankAccountCustomer(sbCustomer);
    }
    for (ForexTradingCustomer fxCustomer : forexTradingCustomers) {
        visitor.visitForexTradingCustomer(fxCustomer);
    }
}
```

## Option B : Step 2 → Add accept(visitor) methods in the customer classes

```
interface VisitableCustomer {
    public void accept(CustomerVisitor visitor);
}
```

```
class HomeLoanCustomer implements VisitableCustomer{
    public void accept(CustomerVisitor visitor) {
        visitor.visitHomeLoanCustomer(this);
    }
    // other related attributes & methods of Home Loan Customer
```

```
class SavingsBankAccountCustomer implements VisitableCustomer{
    public void accept(CustomerVisitor visitor) {
        visitor.visitSavingsBankAccountCustomer(this);
    }
    // other related attributes & methods of Savings Bank Account Customer
```

```
class DematCustomer implements VisitableCustomer  {
    public void accept(CustomerVisitor visitor) {
        visitor.visitDematCustomer(this);
    }
    // other related attributes & methods related to Demat Customer
```

```
class ForexTradingCustomer implements VisitableCustomer {
    public void accept(CustomerVisitor visitor) {
        visitor.visitForexTradingCustomer(this);
    }
    // other related attributes & methods of Forex Trading
```



## Option B : Step 3 → Client pass visitor object to list of Element

88

```
// master list of HomeLoanCustomers, DematCustomers,
// SavingsBankAccountCustomers and ForexTradingCustomers
List<VisitableCustomer> allCustomers = new ArrayList<VisitableCustomer>();

List<CustomerVisitor> visitors = new ArrayList<CustomerVisitor>();
visitors.add(new PrintPhotoId());
visitors.add(new PrintNewYearGreetings());
visitors.add(new PrintDiscountCoupons());
//optionB
for (CustomerVisitor visitor : visitors) {
    for (VisitableCustomer customer : allCustomers) {
        customer.accept(visitor);
    }
}
```

# Visitor Pattern



## Intent

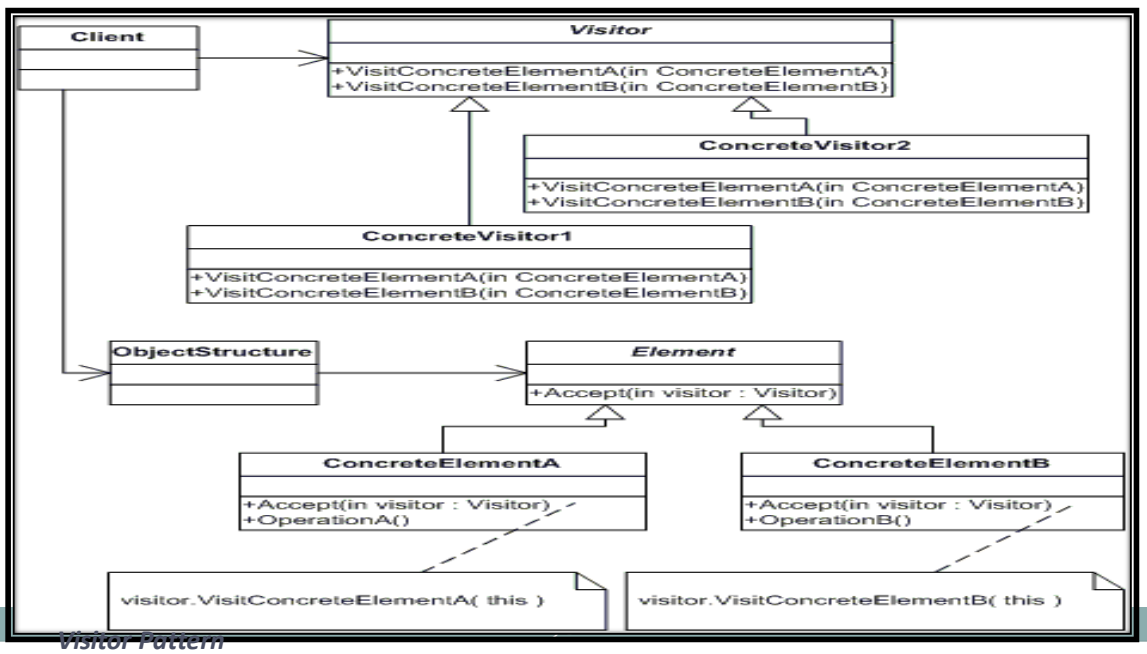
- Represents an operation to be performed on the elements of an object structure.
- Visitor lets you define a new operation without changing the classes of the elements on which it operates.

## Applicability

- Use the Visitor pattern to support many distinct and unrelated operations on objects in a data structure, without burdening their classes with such operations



# Structure



# When and How to implement Visitor

91

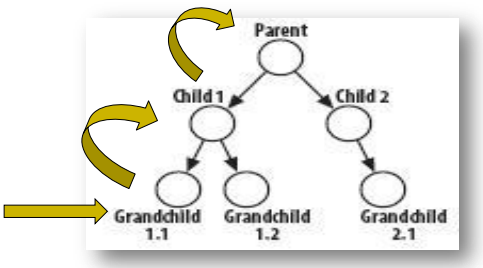
- Confirm that the current hierarchy (known as the Element hierarchy) will be fairly stable and that the public interface of these classes is sufficient for the access the Visitor classes will require. If these conditions are not met, then the Visitor pattern is not a good match.
- Create a Visitor base class with a visit(ElementXxx) method for each Element derived type.
- Add an accept(Visitor) method to the Element hierarchy. The implementation in each Element derived class is always the same – `accept(Visitor v) { v.visit( this ); }` Because of cyclic dependencies, the declaration of the Element and Visitor classes will need to be interleaved.
- The Element hierarchy is coupled only to the Visitor base class, but the Visitor hierarchy is coupled to each Element derived class. If the stability of the Element hierarchy is low, and the stability of the Visitor hierarchy is high; consider swapping the 'roles' of the two hierarchies.
- Create a Visitor derived class for each "operation" to be performed on Element objects. visit() implementations will rely on the Element's public interface.
- The client creates Visitor objects and passes each to Element objects by calling accept().



# Pros and Cons

93

- Ease of adding new features
- Allows better segregation of operations.
- Related operations can be applied on unrelated classes
- Minimal code duplication
- Visitors can accumulate state as they visit each element in the object structure.
- Adding a new ConcreteElement v/s adding a new ConcreteVisitor.




Iterator

## Problem Statement and Intent

95

- Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

**Intent**

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Make it possible to decouple collection classes and algorithms.
- Polymorphic traversal

# Iterator

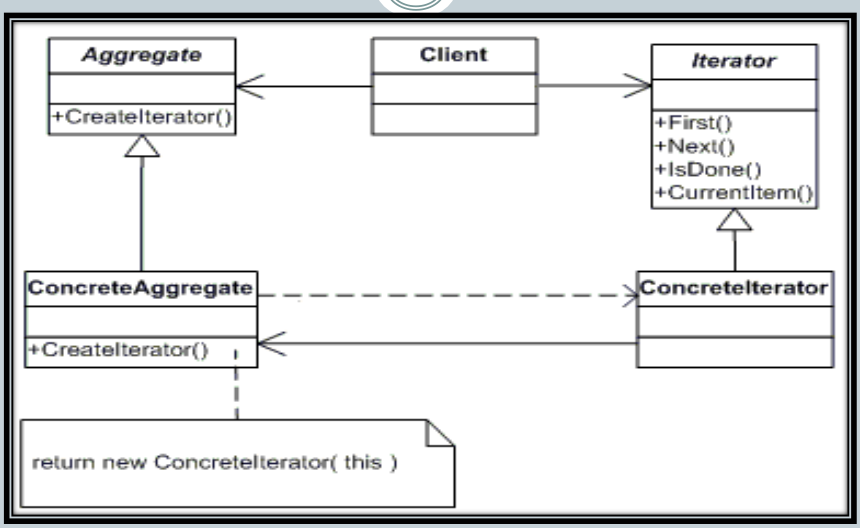
96

- Used for providing a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



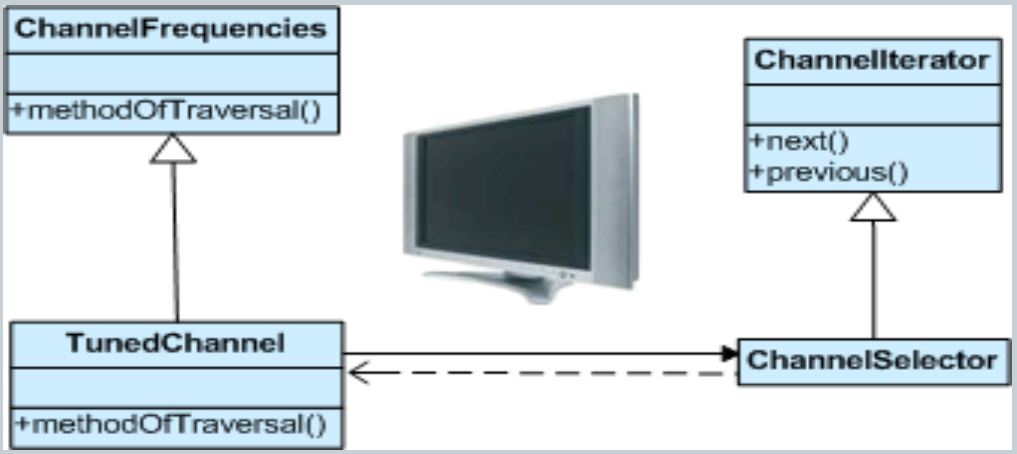
# Structure

97



# Example

98






# How to use Iterator?

99

- Add a returnIterator() method to the "collection" class, and grant the "iterator" class privileged access.
- Design an "iterator" class that can encapsulate traversal of the "collection" class.
- Clients ask the collection object to create an iterator object.
- Clients use the first(), is_done(), next(), and current_item() protocol to access the elements of the collection class.

## Iterator in Java & C#

100

- java.util.Iterator<T> interface in java
- System.Collections.Generic.IEnumerator<T> in C#
- In the foreach function in both java and C#

Mediator

# Mediator



- Define an object that encapsulates how a set of objects interact. Reduce coupling between objects by keeping them from referring to each other explicitly.



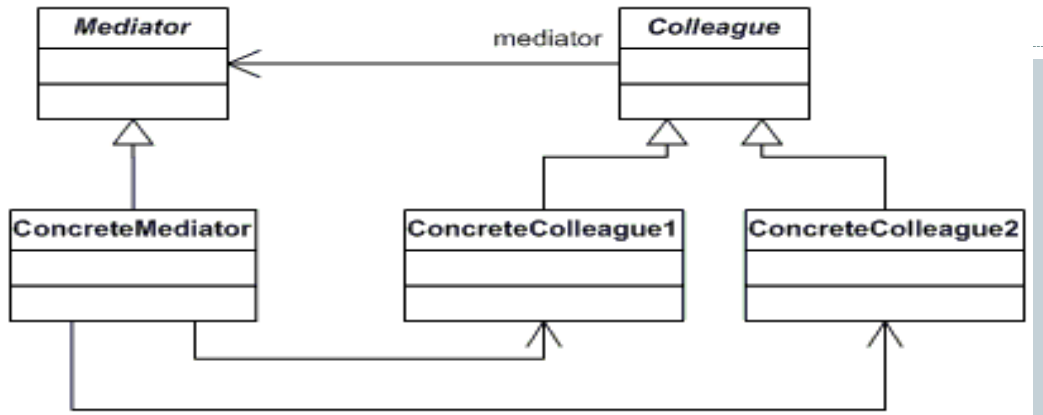
# Applicability



Use the Mediator pattern when

- A set of objects communicates in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- Reusing an object is difficult because it refers to and communicates with many other objects.
- A behavior that is distributed between several classes should be customizable without a lot of sub classing.

# Structure

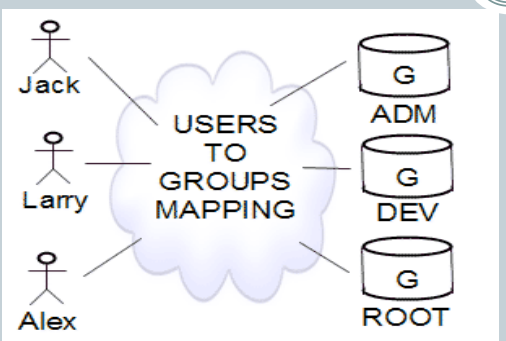


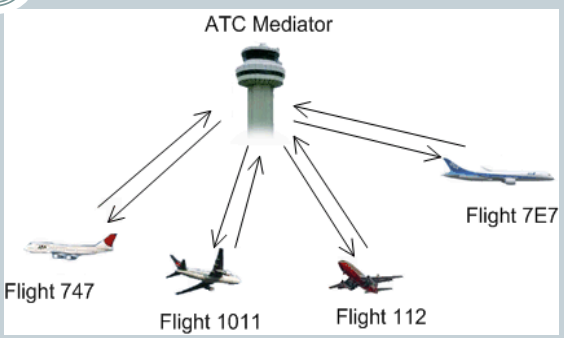

**Mediator (IChatroom)**
defines an interface for communicating with Colleague objects
**ConcreteMediator (Chatroom)**
implements cooperative behavior by coordinating Colleague objects
knows and maintains its colleagues
**Colleague classes (Participant)**
each Colleague class knows its Mediator object
each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague



# Other examples





# Consequences



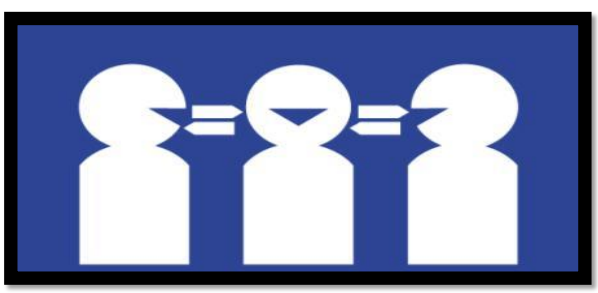
**Advantages**

- **Comprehension** - The mediator encapsulate the logic of mediation between the colleagues. From this reason it' more easier to understand this logic since it is kept in only one class.
- **Decoupled Colleagues** - The colleague classes are totally decoupled. Adding a new colleague class is very easy due to this decoupling level.
- **Simplified object protocols** - The colleague objects need to communicate only with the mediator objects. Practically the mediator pattern reduce the required communication channels(protocols) from many to many to one to many and many to one.
- **Limits Subclassing** - Because the entire communication logic is encapsulated by the mediator class, when this logic need to be extended only the mediator class need to be extended.

**Disadvantages**

- **Complexity** - in practice the mediators tends to become more complex and complex. A good practice is to take care to make the mediator classes responsible only for the communication part. For example when implementing different screens the the screen class should not contain code which is not a part of the screen operations. It should be put in some other classes.





Interpreter

# Interpreter



- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- The implementation of the Interpreter pattern is just the use of the composite pattern applied to represent a grammar. The Interpreter defines the behavior while the composite defines only the structure.



# Boolean expression



- (a and true) or (b and not a)
- (x and y) or true
- y and not (x and not z)

## Steps in interpreting a language

110

- Parsing the language symbols into tokens and syntax checking
- Converting the tokens into a suitable data structure
- Executing the actions as per the data structure



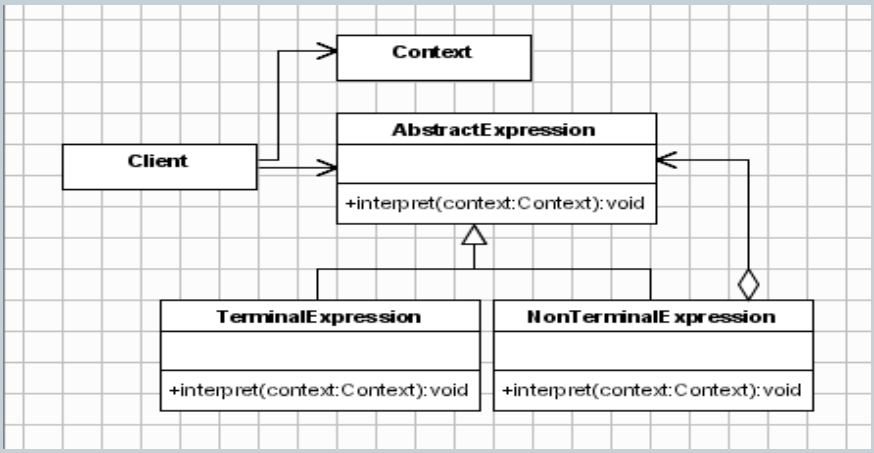
## Applicability

111

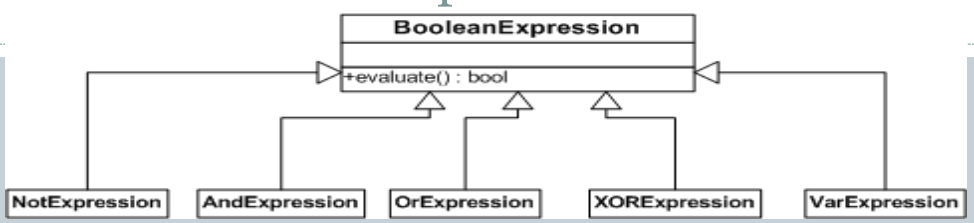
- The Interpreter pattern is used exhaustively in defining grammars, tokenize input and store it.
- A specific area where Interpreter can be used are the rules engines.
- The Interpreter pattern can be used to add functionality to the composite pattern.

# Structure






# Interpreter

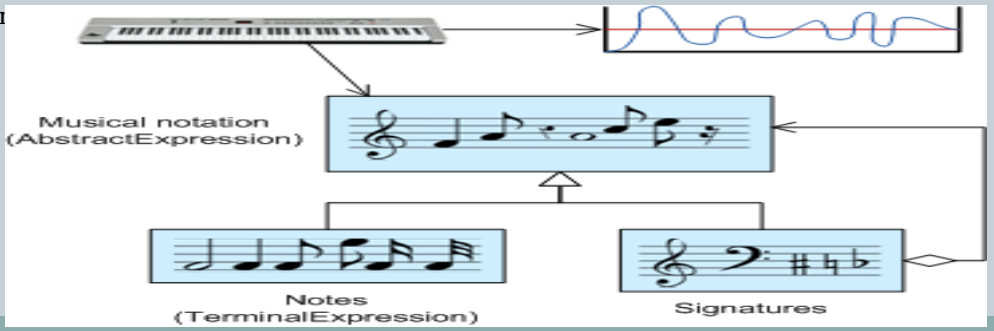



```
(a and true) or (b and not a) is interpreted as
BooleanExpression exp = new OrExpression(new
  AndExpression(newVarExpression("a"), new
  ConstExpression(true)),
  new AndExpression(new VarExpression("b"), new
  NotExpression(new VarExpression("a"))));
```

# Applicability



- The Intepreter pattern defines a grammatical representation for a language and an interpreter to interpret the grammar.
- Use the Interpreter pattern when there is a language to interpret and you can represent statements in the language as abstract syntax trees. The interpreter pattern works best when the
  1. Grammar is simple
  2. Efficien




*Interpreter Pattern*

# Example – Roman numerals converter



- **ThousandExpression, HundredExpression, TenExpression, OneExpression all (TerminalExpression)** - Those classes are usued to define each specific expression. Usually, the TerminalExpression classes implement the interpret method. In our case the method is already defined in the base Expression class and each TerminalExpression class defines its behaviour by implmenting the abstract methods: one, four(), five(), nine(), multiplier(). It is a template method pattern

# Code for Roman numeral to Integer



```
class Context {
    private String input;
    private int output;
    public Context(String input) {
        this.input = input;
    }
    public String getInput() {
    public void setInput(String input) {
    public int getOutput() {
    public void setOutput(int output) {
}
```

```
abstract class Expression {
    public void interpret(Context context) {
        if (context.getInput().length() == 0)
            return;
        if (context.getInput().startsWith(nine())) {
            context.setOutput(context.getOutput() + (9 * multiplier()));
            context.setInput(context.getInput().substring(2));
        } else if (context.getInput().startsWith(four())) {
            context.setOutput(context.getOutput() + (4 * multiplier()));
            context.setInput(context.getInput().substring(2));
        } else if (context.getInput().startsWith(five())) {
            context.setOutput(context.getOutput() + (5 * multiplier()));
            context.setInput(context.getInput().substring(1));
        }
        while (context.getInput().startsWith(one())) {
            context.setOutput(context.getOutput() + (1 * multiplier()));
            context.setInput(context.getInput().substring(1));
        }
    }
    public abstract String one();
    public abstract String four();
    public abstract String five();
    public abstract String nine();
    public abstract int multiplier();
```

```
class ThousandExpression extends Expression {
    public String one() {
        return "M";
    }
    public String four() {
        return " ";
    }
    public String five() {
        return " ";
    }
    public String nine() {
        return " ";
    }
    public int multiplier() {
        return 1000;
    }
}
```

```
class HundredExpression extends Expression {
    public String one() {
        return "C";
    }
    public String four() {
        return "CD";
    }
    public String five() {
        return "D";
    }
    public String nine() {
        return "CM";
    }
    public int multiplier() {
        return 100;
    }
```

```
class TenExpression extends Expression {
    public String one() {
        return "X";
    }
    public String four() {
        return "XL";
    }
    public String five() {
        return "L";
    }
    public String nine() {
        return "XC";
    }
    public int multiplier() {
        return 10;
    }
```

```
class OneExpression extends Expression {
    public String one() {
        return "I";
    }
    public String four() {
        return "IV";
    }
    public String five() {
        return "V";
    }
    public String nine() {
        return "IX";
    }
    public int multiplier() {
        return 1;
    }
}
```

```
public class MainInterpreter {

    public static void main(String[] args) {
        String roman = "MCMXXVIII";
        Context context = new Context(roman);
        // Build the Parse tree
        ArrayList<Expression> expressionlist = new ArrayList<Expression>();
        expressionlist.add(new ThousandExpression());
        expressionlist.add(new HundredExpression());
        expressionlist.add(new TenExpression());
        expressionlist.add(new OneExpression());
        // Interpret
        for (Iterator it = expressionlist.iterator(); it.hasNext();) {
            Expression exp = (Expression) it.next();
            exp.interpret(context);
        }
        System.out.println(roman + " = "
                + Integer.toString(context.getOutput()));
    }
}
```

```
Markers | Properties | Servers | Data Source Explorer | Snippets | Console | XPath
<terminated> MainInterpreter [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (08-Jun-2013 8:38:07 AM)
MCMXXVIII = 1928
```

# Consequences



- It is easy to change and extend the grammar
- Implementing the grammar is easy
- Complex grammars are hard to maintain
- Adding new ways to interpret expressions is easy. For example, you can support formatted printing or type-checking an expression by defining a new operation on the expression class.

| Creational Patterns | |
|---|---|
| Abstract Factory | Creates an instance of several families of classes |
| Builder | Separates object construction from its representation |
| Factory Method | Creates an instance of several derived classes |
| Prototype | A fully initialized instance to be copied or cloned |
| Singleton | A class of which only a single instance can exist |
| **Structural Patterns** | |
| Adapter | Match interfaces of different classes |
| Bridge | Separates an object's interface from its implementation |
| Composite | A tree structure of simple and composite objects |
| Decorator | Add responsibilities to objects dynamically |
| Facade | A single class that represents an entire subsystem |
| Flyweight | A fine-grained instance used for efficient sharing |
| Proxy | An object representing another object |
| **Behavioral Patterns** | |
| Chain of Resp. | A way of passing a request between a chain of objects |
| Command | Encapsulate a command request as an object |
| Interpreter | A way to include language elements in a program |
| Iterator | Sequentially access the elements of a collection |
| Mediator | Defines simplified communication between classes |
| Memento | Capture and restore an object's internal state |
| Observer | A way of notifying change to a number of classes |
| State | Alter an object's behavior when its state changes |
| Strategy | Encapsulates an algorithm inside a class |
| Template Method | Defer the exact steps of an algorithm to a subclass |
| Visitor | Defines a new operation to a class without change |



# References




- GoF Patterns
  - James W Cooper. The design Patterns Java companion
  - Erich Gamma et al. Design Patterns : Elements of Reusable Object Oriented Software
  - Mark Grand Patterns in Java, volume 1.
  - Partha Kuchana. Software Architecture Design Patterns in Java
- UML
  - Grady Booch et al. The Unified Modeling Language User Guide
  - Martin Fowler – UML Distilled
- Websites
  - *www.omg.org* for formal UML specifications
  - *www.knowledgeexchanger.com* for Design Patterns
  - *www.softpatterns.com/*